OpenMP Programming: Correctness, Tuning, Examples

Reinhold Bader (LRZ), Georg Hager (RRZE), Gerhard Wellein (RRZE), Markus Wittmann (RRZE)
Telling the Intel compiler and runtime what to do

- **Compiler options**
  - `-qopenmp`: activate OpenMP directives. Note that this implies `-auto` (all variables of intrinsic type go to the stack)
  - `-qopenmp-stubs`: use stub library, produce standard sequential code
  - `-qopenmp-link=[static|dynamic]`: link OpenMP lib statically or dynamically (default is `dynamic`)
  - `-qopenmp-simd`: turn on OpenMP SIMD compilation independent of OpenMP features (default off, when no OpenMP enabled, on when OpenMP enabled)

- **Run time options**
  - **stack size**: If you get segmentation faults, the following options are available:
    1. `ulimit -s 1000000`
       (increase stack limit after checking)
    2. replace large automatic arrays by allocatables
    3. increase the thread-private stack size from default 4m. e.g.:
       ```
       export KMP_STACKSIZE=2g
       ```
       or (OpenMP 3.0):
       ```
       export OMP_STACKSIZE=2g
       ```
Generate optimization reports with the Intel compiler

- **-qopt-report[=0-5]**
genenerate optimization report and control the detail of the report (default is 2)

- **-qopt-report-phase[=(all|openmp|vec)...]**
produces further diagnostic information (default is all)

- **-qopt-report-annotate[=(text|html)]**
produce an annotated source listing as text or HTML (default is text)

- **-qopt-report-routine=<string>**
generate report only for routines containing <string> in their name

- **-qopt-report-filter=<string>**
double quote <string>, allowed formats of <string>:
  - filename
  - filename,routine
  - filename,linestart-lineend,…
  - filename,routine,linestart-lineend,…
  - can be concatenated via semicolon
  - -qopt-report-filter="test.F90,’copy’"
  - -qopt-report-filter="test.F90,10-20;test.F90,’copy’"
General and Intel-specific run-time options for OpenMP

- **OMP_* variables**
  - these have been discussed in the OpenMP talks

- **Intel-specific settings:**
  - **KMP_LIBRARY**: OpenMP execution mode can be set to one of the following:
    - **throughput**: threads will be put to sleep after a waiting time determined by the `KMP_BLOCKTIME` environment variable. Useful if resources are shared. This is the **default** setting
    - **turnaround**: worker threads stay active all the time, do not yield for other threads. Useful if dedicated resources available
    - **serial**: forces serial execution
Intel runtime (cont’d)

- **KMP_BLOCKTIME**: set to the waiting time for throughput mode. In units of milliseconds. Default is 200.
  - other units are possible, e.g. `KMP_BLOCKTIME=4s` specifies 4 seconds

- **KMP_ALL_THREADS**: maximum number of threads allowed in a parallel region. Default is essentially 4 * (number of CPUs)

- **KMP_*** library routines for performing settings and thread-local memory management at run time.
  - disadvantage: results in (potentially) non-portable programs

- **KMP_DETERMINISTIC_REDUCTION**: identical floating point reductions

- **KMP_SETTINGS**: display all (KMP/OMP) settings

- **KMP_AFFINITY**: bind threads to logical processors / cores / sockets; possible values:
  - `compact,<level>` (level=0,1,2)
  - relies on thread binding facilities in OS
  - `verbose` shows detailed information
  - Round-robin to physical cores: `KMP_AFFINITY=verbose, granularity=fine, compact,1,0`
KMP_AFFINITY example

```
$ env KMP_AFFINITY="verbose,granularity=fine,compact,1,0" OMP_NUM_THREADS=4 <openmp-binary>
OMP: Info #204: KMP_AFFINITY: decoding x2APIC ids.
OMP: Info #202: KMP_AFFINITY: Affinity capable, using global cpuid leaf 11 info
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected: 
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31}
OMP: Info #156: KMP_AFFINITY: 32 available OS procs
OMP: Info #157: KMP_AFFINITY: Uniform topology
OMP: Info #179: KMP_AFFINITY: 2 packages x 8 cores/pkg x 2 threads/core (16 total cores)
OMP: Info #206: KMP_AFFINITY: OS proc to physical thread map:
OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to package 0 core 0 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 16 maps to package 0 core 0 thread 1
...
OMP: Info #242: KMP_AFFINITY: pid 16822 thread 0 bound to OS proc set {0}
OMP: Info #242: KMP_AFFINITY: pid 16822 thread 1 bound to OS proc set {1}
OMP: Info #242: KMP_AFFINITY: pid 16822 thread 2 bound to OS proc set {2}
OMP: Info #242: KMP_AFFINITY: pid 16822 thread 3 bound to OS proc set {3}
...
```

... Application output
Correctness of OpenMP programs
Some pitfalls using OpenMP

- OpenMP is easier to handle
  - no explicit mapping of data to processors necessary
- But: possible resource conflicts
  - incorrectly programmed access to shared resource
- Major classes of errors:
  - race condition: results of program depend on detailed timing of threads → usually incorrect
  - sharing attributes: default sharing attributes
  - deadlock: (some) threads lock up waiting for a resource which is never freed
Two or more threads may concurrently access uncoordinatedly the same shared variable

Occurrence is nondeterministic
  - Scheduling of threads
  - Number of threads
  - Load on the system
  - Input data

```c
#pragma omp parallel for
for (i = 0; i < n - 1; ++i) {
    a[i] = a[i] + b[i];
}
```

```c
#pragma omp parallel for
for (i = 0; i < n - 1; ++i) {
    a[i] = a[i + 1] + b[i];
}
```

Data Race through loop carried dependency
OpenMP errors: race conditions II

- Use suitable synchronization construct to fix
  - may effectively serialize program!

```c
ic = 0
!$omp parallel sections
!$omp section
  a = b + c
!$omp section
  b = a + c
!$omp section
  c = b + a
!$omp end parallel sections
```

- may produce wrong answers
- varying from run to run and with number of threads
OpenMP errors: sharing attributes I

- Default data sharing attribute is **shared**
- There exist some exceptions...

```c
#pragma omp parallel for
for (i=0; i<n; ++i) {
    ... 
}

Loop counter variables become implicitly private.

```c
#pragma omp parallel for
for (i=0; i<n; ++i) {
    for (j=0; j<n; ++j) {
        ...
    }
} } i private, j shared
Only loop counters of associated loops become private.

```c
#pragma omp parallel for
for (i=0, j=0; i<n; ++i, ++j) {
    ...
} } init increment
Not conforming with the standard. Only one init expression and one increment expression are allowed.

```c
#pragma omp parallel for collapse(2)
for (i=0; i<n; ++i) {
    for (j=0; j<n; ++j) {
        ...
    }
} } i private, j private
Collapse associates “collapsed” loops with the loop construct.
```
more with Fortran…

```fortran
!$omp parallel do
do i = 1, n
  do j = 1, n
    ...
    end do
  end do
end do
```

- **Implicitly** `i` and `j` become private.
- **Fortran only:**
  - Any of the loop iteration variables are made private.
Subroutine dot(n, a, b, c)
  implicit none
  integer :: n, i
  integer :: a(1:n), b(1:n), c, cl

!$omp parallel shared(n,a,b,c)
!!omp& private(i,cl)
!$omp do
  do i = 1, n
    cl = cl + b(i) * a(i)
  end do
!$omp end do
!$omp critical
  c = c + cl
!$omp end parallel

return
end subroutine dot

Spot the error!

- Continuation of OpenMP directive treated as comment, because of extra “!”.
  - !$omp& would be correct
- Variable i becomes implicitly private → OK
- Variable cl becomes shared.
- Variable cl is no initialized inside the parallel region.
- With default(none) the compiler would recognize the error.
```c
void compute(int n) {
    int i;
    double h, x, sum;

    h = 1.0 / (double)n;
    sum = 0.0;

    #pragma omp for reduction(+:sum) shared(h)
    for (i=1; i<=n; ++i) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x * x));
    }

    pi = h * sum;
}
```

- Variable `i` is implicitly private. → OK
- Variable `x` should be private, but is shared by default, which leads to a data race.
call omp_init_lock(var)
!$omp parallel sections
!$omp section
  call omp_set_lock(var)
  ... (work)
if (ival > tol) then
  call omp_unset_lock(var)
else
  call error(ival)
end if
!$omp section
  call omp_set_lock(var)
  ... (work2 depends on work)
call omp_unset_lock(var)
!$omp end parallel sections

- Typical situation
  - using locks
  - violation of well defined sequence of OpenMP calls

*var* potentially never unlocked
Example
Parallel Histogram Computation
The problem

- Compute simplified histogram of a (integer) random number generator:
  \[ \text{hist}[\text{rand()} \mod 16] \]
- Check if \text{rand()} generates a homogeneous distribution:
  \[ \text{hist}[\text{rand()} \mod 16] = N/16 \]
  (N: random numbers generated)

- Architecture: Intel Xeon/Sandy Bridge 2.7 GHz (fixed clock speed)
- Compiler: Intel 14.0 (no inlining)
- Simple Random number generator (taken from man page of \text{rand}; there are much better ones…)

```c
int myrand(unsigned long* next)
{
    *next = *next * 1103515245 + 12345;
    return((unsigned)(*next/65536) % 32768);
}
```
Serial implementation and baseline

**Computation**

```c
lseed = 123;
for(i = 0; i < 16; ++i)
    hist[i] = 0;

timing(&wcstart, &ct);

for(i = 0; i < N; ++i)
    hist[RAND & 0xf]++;

timing(&wcend, &ct);
```

**Quality evaluation**

```c
double av = N / 16.0;
double maxerr = 0.0;
double err;

for(i = 0; i < 16; ++i) {
    err = (hist[i] - av) / av;
    maxerr = MAX(fabs(err), maxerr);
}
```

- **Serial baselines (N=10^9)**

  - RAND = myrand(&lseed)
    - Time = 3.6 s
    - maxerr = 3 * 10^{-6}

  - RAND = rand_r(&lseed)
    - Time = 6.7 s
    - maxerr = 4 * 10^{-6}
### Straightforward parallelization?!

- Just add a single OpenMP directive.....

```c
lseed = 123;
for(i = 0; i < 16; ++i)
  hist[i] = 0;

timing(&wcstart, &ct);

#pragma omp parallel for
for(i = 0; i < N; ++i) {
  hist[myrand(&lseed) & 0xf]++;
}

timing(&wcend, &ct);
```

#### Result Quality

<table>
<thead>
<tr>
<th>Threads</th>
<th>maxerr</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>~0.47</td>
</tr>
<tr>
<td>4</td>
<td>~0.72</td>
</tr>
<tr>
<td>8</td>
<td>~0.86</td>
</tr>
<tr>
<td>16</td>
<td>~0.89</td>
</tr>
</tbody>
</table>

#### Performance

<table>
<thead>
<tr>
<th>Threads</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>~20s</td>
</tr>
<tr>
<td>4</td>
<td>~23s</td>
</tr>
<tr>
<td>8</td>
<td>~28s</td>
</tr>
<tr>
<td>16</td>
<td>~105s</td>
</tr>
</tbody>
</table>

**Problem**: Uncoordinated concurrent updates of `hist[]` and `lseed`  
→ Runtime and result changes between runs
Get it correct first!

- Protect update of lseed and hist[] by critical region

Result Quality

<table>
<thead>
<tr>
<th>Threads</th>
<th>maxerr</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$3 \times 10^{-6}$</td>
</tr>
<tr>
<td>4</td>
<td>$3 \times 10^{-6}$</td>
</tr>
<tr>
<td>8</td>
<td>$3 \times 10^{-6}$</td>
</tr>
<tr>
<td>16</td>
<td>$3 \times 10^{-6}$</td>
</tr>
</tbody>
</table>

Performance

<table>
<thead>
<tr>
<th>Threads</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>201s</td>
</tr>
<tr>
<td>4</td>
<td>221s</td>
</tr>
<tr>
<td>8</td>
<td>217s</td>
</tr>
<tr>
<td>16</td>
<td>427s</td>
</tr>
</tbody>
</table>

Baseline: 3*10^{-6} Baseline: 3.6s

```c
#pragma omp parallel for
for(i = 0; i < N; ++i) {

    #pragma omp critical
    hist[myrand(&lseed) & 0xf]++;
}
```

Result Quality: OK

**Problem**: Performance: ~50x-100x slower! Serialization and some (?) more overhead, e.g. “synchronization”

© 2014 - 2018 LRZ and RRZE
Avoid complete serialization

- Define a private `lseed` and `value`
- Only histogram update needs a `#pragma omp critical`

Result Quality

<table>
<thead>
<tr>
<th>Threads</th>
<th>maxerr</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$6 \times 10^{-6}$</td>
</tr>
<tr>
<td>4</td>
<td>$15 \times 10^{-6}$</td>
</tr>
<tr>
<td>8</td>
<td>$24 \times 10^{-6}$</td>
</tr>
<tr>
<td>16</td>
<td>$60 \times 10^{-6}$</td>
</tr>
</tbody>
</table>

```
#pragma omp parallel for \
    firstprivate(lseed) private(value)
for(i = 0; i < N; ++i) {
    value = myrand(&lseed) & 0xf;
    #pragma omp critical
    hist[value]++;
}
```

Performance

<table>
<thead>
<tr>
<th>Threads</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>191s</td>
</tr>
<tr>
<td>4</td>
<td>201s</td>
</tr>
<tr>
<td>8</td>
<td>194s</td>
</tr>
<tr>
<td>16</td>
<td>413s</td>
</tr>
</tbody>
</table>

Baseline: 3.6s

Problem: Performance improves only marginally → critical is still an issue!

Problem (?): Result Quality is slightly worse than baseline.
Get rid of the critical statement (1)

Use a shared scoreboard (hist_2D):

• Each thread writes to a separate column of length 16

• Sum up the numbers across each row to get the final hist[]

```c
// additional shared array
// assuming 4 threads
hist_2D[16][4] = { 0 };

#pragma omp parallel
{
    int tId = omp_get_thread_num();

    #pragma omp for \
    firstprivate(lseed) private(value)
    for(i = 0; i < N; ++i) {
        value = myrand(&lseed) & 0xf;
        hist_2D[value][tId]++;
    }

    #pragma omp critical
    for (i = 0; i < 16; ++i)
    hist[i] += hist_2D[i][tId];
}
```

Use a shared scoreboard (hist_2D):

```
[0,0] [0,1] [0,2] [0,3]
[1,0] [1,1] [1,2] [1,3]
... ... ... ...
[14,0] [14,1] [14,2] [14,3]
[15,0] [15,1] [15,2] [15,3]
```

```
[0] [1] [14] [15]
```

```
histogram_omp_d
```
Get rid of the critical statement (2)

### Result Quality

<table>
<thead>
<tr>
<th>Threads</th>
<th>maxerr</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$6 \times 10^{-6}$</td>
</tr>
<tr>
<td>4</td>
<td>$15 \times 10^{-6}$</td>
</tr>
<tr>
<td>8</td>
<td>$24 \times 10^{-6}$</td>
</tr>
<tr>
<td>16</td>
<td>$60 \times 10^{-6}$</td>
</tr>
</tbody>
</table>

### Performance

<table>
<thead>
<tr>
<th>Threads</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>11.7s</td>
</tr>
<tr>
<td>4</td>
<td>9.3s</td>
</tr>
<tr>
<td>8</td>
<td>6.6s</td>
</tr>
<tr>
<td>16</td>
<td>19.3s</td>
</tr>
</tbody>
</table>

Performance improves 30x but still much slower than serial version?!

Each thread writes frequently to every cache line of `hist_2D` → False Sharing
Avoid False Sharing

Use thread private histogram (\texttt{hist\_local[16]}) for thread local computation & sum up all results at the end

```
#pragma omp parallel
{
  int hist_local[16] = { 0 };

  // first private
  #pragma omp for \firstprivate(lseed) private(value)
  for(i = 0; i < N; ++i) {
    value = myrand(&lseed) & 0xf;
    hist_local[value]++;
  }

  #pragma omp critical
  for (i = 0; i < 16; ++i)
    hist[i] += hist_local[i];
}
```

**Result Quality**

<table>
<thead>
<tr>
<th>Threads</th>
<th>maxerr</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$6 \times 10^{-6}$</td>
</tr>
<tr>
<td>4</td>
<td>$15 \times 10^{-6}$</td>
</tr>
<tr>
<td>8</td>
<td>$24 \times 10^{-6}$</td>
</tr>
<tr>
<td>16</td>
<td>$60 \times 10^{-6}$</td>
</tr>
</tbody>
</table>

**Performance**

<table>
<thead>
<tr>
<th>Threads</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.78s</td>
</tr>
<tr>
<td>4</td>
<td>0.89s</td>
</tr>
<tr>
<td>8</td>
<td>0.44s</td>
</tr>
<tr>
<td>16</td>
<td>0.22s</td>
</tr>
</tbody>
</table>

**Baseline:** 3.6s

**Baseline:** 3*10^{-6}

**PROBLEM:** Quality still gets worse as number of threads increase?!

Every thread does the same (\texttt{lseed} is the same!)

-> more threads less statistics

**Performance:** OK now – nice scaling
Use different seeds for each thread!

### Result Quality

<table>
<thead>
<tr>
<th>Threads</th>
<th>maxerr</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$4 \times 10^{-6}$</td>
</tr>
<tr>
<td>4</td>
<td>$7 \times 10^{-6}$</td>
</tr>
<tr>
<td>8</td>
<td>$10 \times 10^{-6}$</td>
</tr>
<tr>
<td>16</td>
<td>$10 \times 10^{-6}$</td>
</tr>
</tbody>
</table>

### Performance

<table>
<thead>
<tr>
<th>Threads</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.78s</td>
</tr>
<tr>
<td>4</td>
<td>0.89s</td>
</tr>
<tr>
<td>8</td>
<td>0.44s</td>
</tr>
<tr>
<td>16</td>
<td>0.22s</td>
</tr>
</tbody>
</table>

Baseline: 3.6s

---

```c
#pragma omp parallel
{
    int hist_local[16] = { 0 };
    int myseed;
    #pragma omp critical
    myseed = myrand(&seed);
    #pragma omp for private(value)
    for(i = 0; i < N; ++i) {
        value = myrand(&myseed) & 0xf;
        hist_local[value]++;
    }
    #pragma omp critical
    for (i = 0; i < 16; ++i)
    {
        hist[i] += hist_local[i];
    }
}
```

Result quality is slightly worse - we are doing different things than in the serial version........
Conclusions from the histogram example

- **Get it correct first!**
  - Race conditions, deadlocks…

- **Avoid complete serialization**
  - Thread-local data

- **Avoid false sharing**
  - Proper shared array layout
  - Padding

- **Parallel random numbers may be non-trivial**